Analyse Syntaxique

1. Syntaxe et grammaire
2. Analyse descendante
3. Analyse ascendante
4. Traitement des erreurs
5. Yacc

Syntaxe

On considère des *langages formels* comme les langages de programmation. On distingue

- La syntaxe : comment on doit écrire.
- La sémantique : ce que ça veut dire.

On utilise pour spécifier la syntaxe des *règles* de grammaire qui ont la forme

symbole → expression

Les expressions utilisent deux sortes de symboles:

1. Ceux du langage final: les symboles *terminaux*
2. Des symboles intermédiaires qui décrivent des *catégories grammaticales*

Définition formelle

Une grammaire est donnée par:

1. Un alphabet A de symboles terminaux.
2. Un ensemble V de symboles non-terminaux.
3. Un ensemble fini de règles
$$(x, w) \in V \times (A + V)^*$$
notées
$$x \to w$$
4. Une variable S appelée axiome.

Un *mot* sur l'alphabet A est une suite d'éléments de A.

On note A^* l'ensemble des mots sur A.

Un langage formel est une partie de A^*. Le langage défini par une grammaire (à partir de S) est l'ensemble

$$L = \{w \in A^* \mid S \xrightarrow{*} w\}$$

notations : $\xrightarrow{*}$ ou $\xRightarrow{*}$, $N = V + A$.

Exemple

Grammaire pour les expressions arithmétiques simples:

$$
\begin{aligned}
E &\rightarrow nombre \\
E &\rightarrow (\, E \,) \\
E &\rightarrow E + E \\
E &\rightarrow E - E \\
E &\rightarrow E * E \\
E &\rightarrow E \,/\, E
\end{aligned}
$$

Le symbole E désigne la catégorie grammaticale des expressions arithmétiques. Nous considérons *nombre* comme un symbole abstrait (destiné à représenter les chaînes de caractères qui sont des nombres).

Les symboles terminaux sont:

$$(\; , \;) \; , \; + \; , \; - \; , \; * \; , \; /$$

Langues naturelles

On peut utiliser les mêmes grammaires pour décrire la syntaxe des langues naturelles. Ainsi on pourra utiliser les règles:

$$
\begin{aligned}
< phrase > &\rightarrow < gsujet >< gverbe > \\
< gsujet > &\rightarrow < det >< adj >< nom > \\
&\quad |< det >< nom > \\
&\quad | \ldots \\
< gverbe > &\rightarrow < verbe >< gcomplement > \\
&\quad |< verbe > \\
&\quad | \ldots
\end{aligned}
$$

Ce sont les grammaires de Chomsky.

Arbre d'analyse

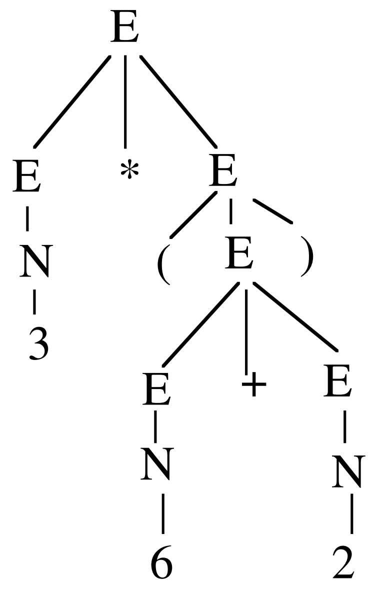


Figure 1: Analyse de $3 * (6 + 2)$

Langage défini par une grammaire : ensemble des frontières d'arbres d'analyse (ce sont des mots sur l'alphabet terminal).

Ambiguïté

Grammaire ambiguë: un même mot possède plusieurs arbres d'analyse.

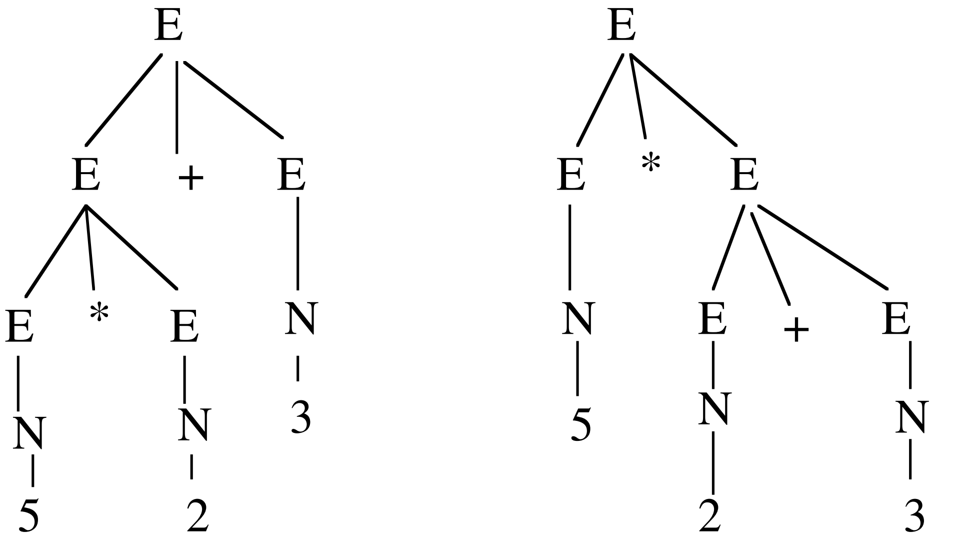


Figure 2: Deux analyses de $5 * 2 + 3$

Exemple classique : la grammaire

$$< inst > \rightarrow \textbf{si} < cond > \textbf{alors} < inst >$$

$$\mid \textbf{si} < cond > \textbf{alors} < inst > \textbf{sinon} < inst >$$

est ambiguë

Exemples

1. La grammaire

$$E \rightarrow E + E \mid N$$

   est ambiguë.

2. La grammaire

$$P \rightarrow (P)P \mid \epsilon$$

   définit les expressions parenthèsées. Elle n'est pas ambiguë.

3. La grammaire

$$E \rightarrow EE+ \mid N$$

   définit les expressions additives en notation suffixe. Elle n'est pas ambiguë.

Une grammaire non-ambiguë pour les expressions

La grammaire E, T, F: on utilise trois catégories au lieu d'une:

1. Facteur: expression qu'on ne peut pas fragmenter.
2. Terme: produit ou quotient de facteurs.
3. Expression: somme ou différence de termes.

$$
\begin{array}{lcl}
E & \to & E + T \mid E - T \mid T \\
T & \to & T * F \mid T/F \mid F \\
F & \to & (E) \mid N
\end{array}
$$

Choix réalisés:

1. Associativité de gauche à droite.
2. Priorité de $*$ et $/$ sur $+$ et $-$

diagrammes de transition

Toute grammaire peut être mise sous la forme de *diagramme de transition*. On aura par exemple :

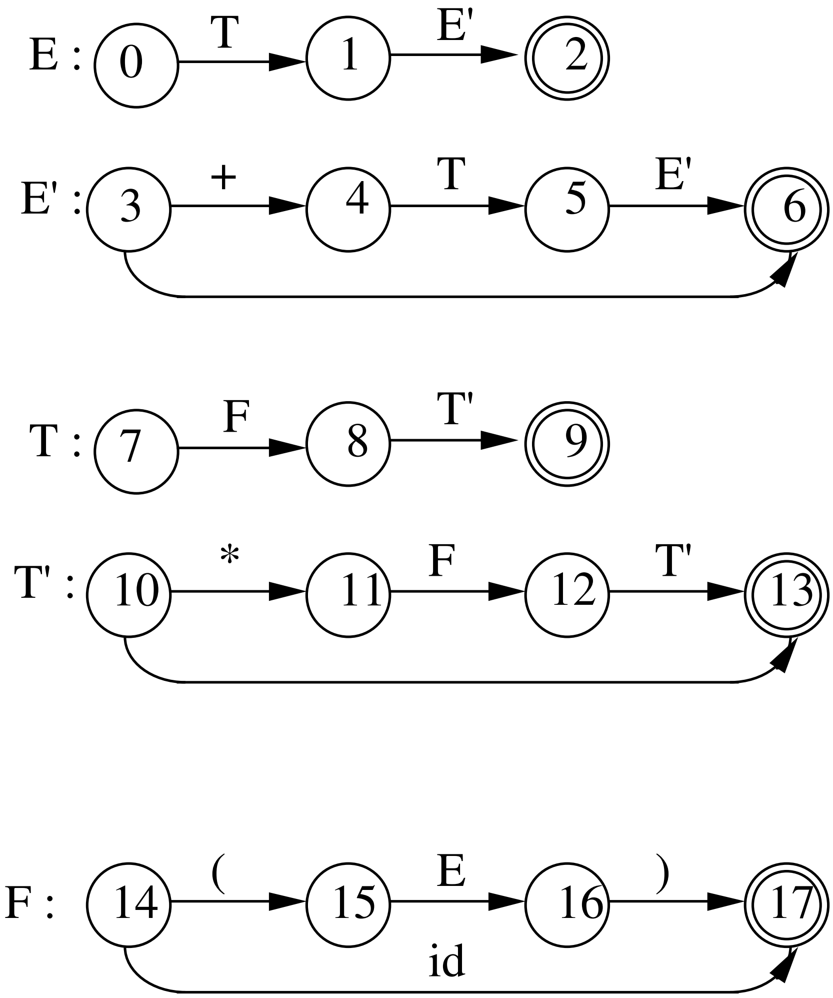
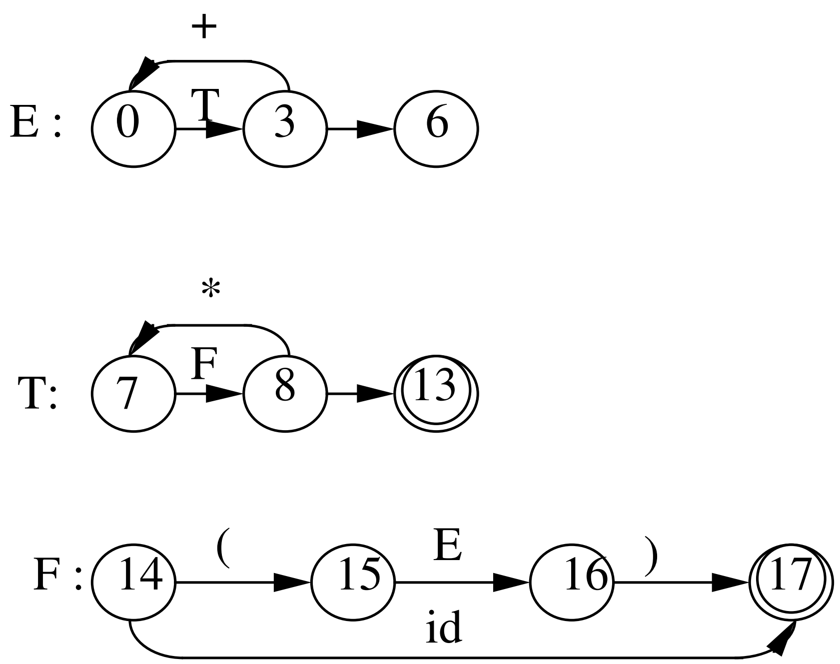


Figure 3: diagrammes

ou sous forme simplifiée :



Analyse syntaxique

Objectif: construction de l'arbre d'analyse d'un mot donné.

Deux méthodes:

- Analyse descendante (=descente recursive): on parcourt la donnée de gauche à droite en appelant des procédures pour chaque symbole de la grammaire.
- Analyse ascendante: on fait le même parcours mais en empilant les expressions identifiées au fur et à mesure.

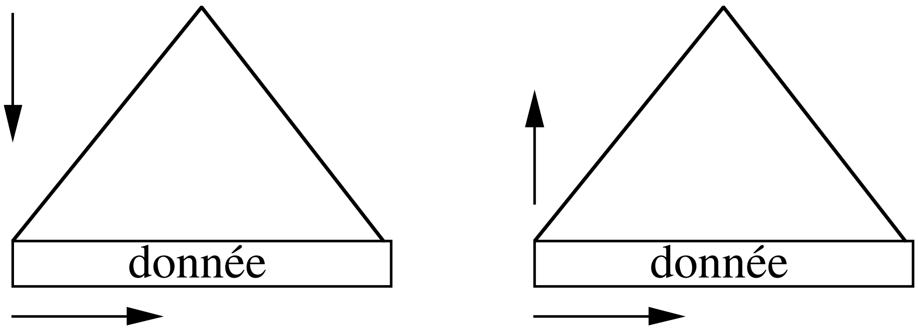


Figure 4: Les deux méthodes

Analyse descendante des expressions arithmétiques

Les fonctions C suivantes effectuent la transformation en notation suffixe d'un expression arithmétique. On part de la grammaire:

$$\begin{array}{rcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid N \end{array}$$

dans laquelle on élimine la *récursivité gauche*:

$$\begin{array}{rcl} E & \to & TE' \\ E' & \to & +TE' \mid \epsilon \\ T & \to & FT' \\ T' & \to & *FT' \mid \epsilon \\ F & \to & (E) \mid N \end{array}$$

On utilise un caractère d'*anticipation* ('look-ahead') pour guider l'analyse. C'est une variable globale de type int.

On écrit d'abord, en associant à E la fonction `Expr`, à T la procedure `Terme` et à E' la procedure `Reste`:

```
void Expr(void){
 Terme(); Reste();
}


void Reste(void){
  if (Symbole == '+'){
    Accepter('+'); Terme();
    putchar('+'); Reste();
  }
}
```

On peut éliminer l'appel récursif terminal dans `Reste` puis recopier `Reste` dans `Expr`. On obtient finalement:

```
void Expr(void){
 Terme();
 while(1)
   if (Symbole == '+'){
     Accepter('+'); Terme(); putchar('+');
   }
   else break;
}
```

En procédant de même pour terme, on obtient :

```
void Terme(void){
  Fact();
  while(1)
   if (Symbole == '*'){
    Accepter('*'); Fact(); putchar('*');
   }
  else break;
}
```

On écrit enfin `Fact` en faisant l'hypothèse simplificatrice que les nombres sont formés d'un seul chiffre.

```
void Fact(void){
  if (Symbole == '('){
    Accepter('('); Expr(); Accepter(')');
  }
  else {
    putchar(Symbole);Accepter(Symbole);
  }
}
```

où Accepter est la fonction :

```
void Accepter(int Lex){
  if (Symbole == Lex)
    Symbole = getchar();
  else Erreur;
}
```

Le programme complet sera:

```
int Symbole;

int main(void){
  Symbole = getchar();
  Expr();
  putchar('\n');
}

void Expr(void){
..
}

void Terme(void){
..
}

void Fact(void){
..
}
voidErreur(void){
  printf("erreur de syntaxe\n");
  exit(1);
}
```

Premier et Suivant

Pour analyser l'algorithme descendant (algorithme LL) on introduit deux fonctions: Premier et Suivant.

Pour chaque variable X, Premier(X) et Suivant(X) sont des ensembles de symboles terminaux.

On a a dans Premier(X) si on peut dériver à partir de X un mot commençant par a.

On a a dans Suivant(X) s'il existe un mot dérivable à partir de l'axiome dans lequel a suit X.

Calcul de Premier et Suivant

Pour calculer les fonctions Premier et Suivant, on construit un graphe pour chacune. Les sommets sont les symboles terminaux ou non. On a un chemin de X à a ssi $a \in \text{Premier}(X)$ (ou $\text{Suivant}(X)$).

Premier

On a une flèche de X vers Y ssi il existe une règle $X \to UYV$ avec $U \xrightarrow{*} \epsilon$ (Y peut être terminal).

Suivant

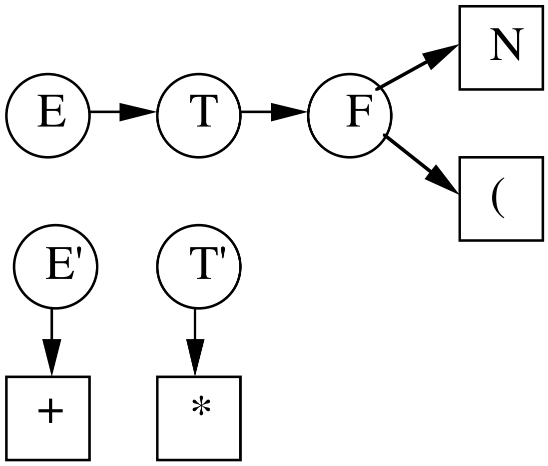
Pour calculer Suivant, on aura une flèche de X à Y dans l'un des deux cas suivants:

1. $Y = a$ et il existe une règle $Z \to UXV$ avec $a \in Premier(V)$
2. il existe une règle $Y \to UXV$ avec $V \xrightarrow{*} \epsilon$ (attention: on remonte les règles)

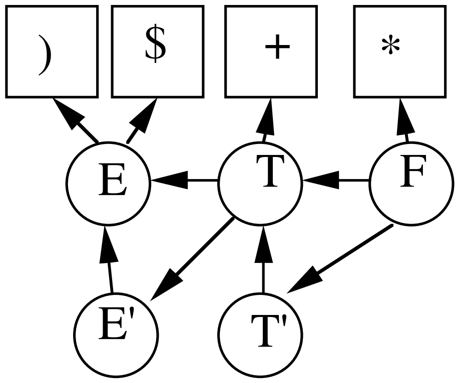
Exemples

Pour la grammaire

$$
\begin{array}{rcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE' \mid \epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT' \mid \epsilon \\
F & \rightarrow & (E) \mid N
\end{array}
$$



Premier



Suivant

Table d'analyse LL

On peut utiliser Premier et Suivant pour construire une table d'analyse de la forme:

	...	a	...
X		n	

où la règle n est ou bien

$$X \to U$$

avec $a \in Premier(U)$ ou bien

$$X \to \epsilon$$

avec $a \in Suivant(X)$.

On dit que la grammaire est LL(1) si on n'obtient pas ainsi de conflits. Ce sont les grammaires pour lesquelles l'analyse descendante (avec une anticipation d'un caractère) est possible.

Exemple

	+	$*$	(	)	N	\$
E			1		1	
E'	2			3		3
T			4		4	
T'	6	5		6		6
F			7		8	

$$
\begin{aligned}
1 : E &\rightarrow TE' \\
2 : E' &\rightarrow +TE' \\
3 : E' &\rightarrow \epsilon \\
4 : T &\rightarrow FT' \\
5 : T' &\rightarrow *FT' \\
6 : T' &\rightarrow \epsilon \\
7 : F &\rightarrow (E) \\
8 : F &\rightarrow N
\end{aligned}
$$

On peut ainsi améliorer l'analyse descendante en traitant les erreurs. La fonction **Expr()** peut s'écrire:

```
Expr()
{
 Terme();
 while(1)
   if (Symbole == '+'){
     Accepter('+'); Terme(); putchar('+');
   }
   else if(Symbole==')'||Symbole=='$')  break;%$
   else error();
}
```

ou encore mieux

```
void Expr(void){
 Terme();
 while(1){
   switch (Symbole){
   case   '+':
       {Accepter('+'); Terme(); putchar('+');break;}
   case ')' : case '$':  return;
   case NUM : case '(' : Erreur("operateur manquant");
   }
 }
```

```
}
```

Analyse ascendante

On utilise une *pile* pour ranger la partie de la donnée qui a déja été réduite.

Il y a deux sorte de mouvements:

1. Empiler de la donnée vers la pile.
2. Réduire un corps de règle en sommet de pile.

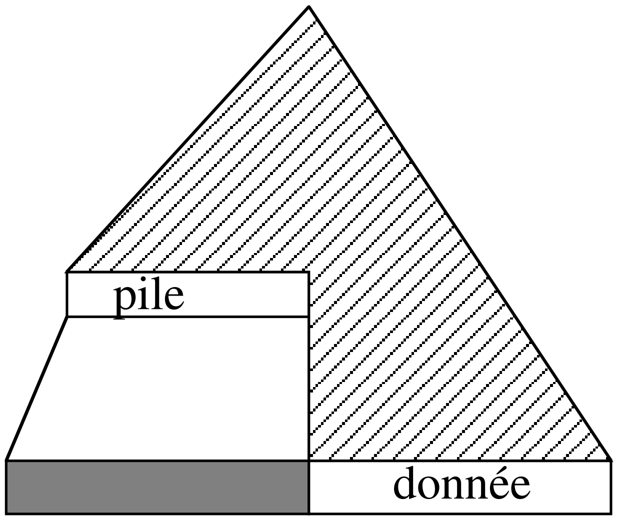


Figure 5: Schéma de fonctionnement

Analyseur LR

On construit un *analyseur* LR utilisant:

1. des *états* donnant l'information sur le contenu de la pile.
2. des *empilements* ajoutant un état en sommet de pile notés ep .
3. des *réductions* par la règle n notées rn
4. des *transitions* donnant l'état après réduction.

ETAT	terminaux	non-terminaux
	a	E
$p : \ldots$	eq	
$q : \ldots$	rn	
$u : \ldots$		v

Si la règle n est $E \to w$, on fait:

1. dépiler $\mid w \mid$ symboles.
2. Si on a l'état u au sommet, empiler v.

Analyse SLR(1)

C'est la stratégie la plus simple. Dans la situation:

	$\ldots$	a	$\ldots$
p		?	

on ne prend en compte une réduction par la règle $X \rightarrow U$ que si $a \in \mathrm{Suivant}(X)$.

Analyse ascendante des expressions arithmétiques

On prend la grammaire:

$$
\begin{aligned}
1: E &\rightarrow E+T \\
2: E &\rightarrow T \\
3: T &\rightarrow T*F \\
4: T &\rightarrow F \\
5: F &\rightarrow (E) \\
6: F &\rightarrow N
\end{aligned}
$$

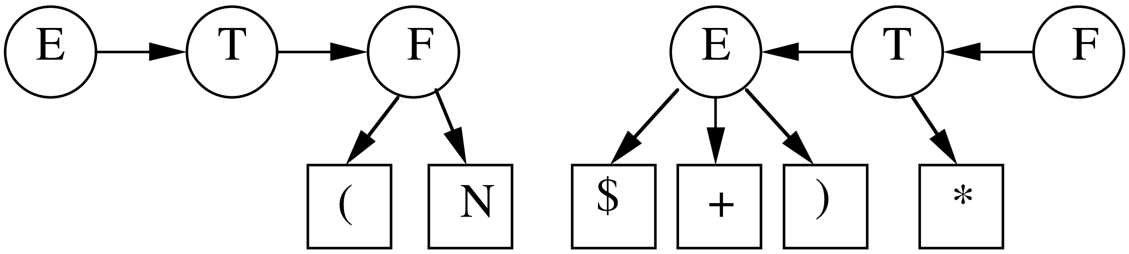


Figure 6: Premier et Suivant

La table d'analyse

ETAT	terminaux						non-termin.		
	N	+	$*$	(	)	\$	E	T	F
0	e5			e4			1	2	3
1: E		e6				acc			
2: T ou $\ldots(T$		r2	**e7**		r2	r2			
3: $\ldots F$		r4	r4		r4	r4			
4: $\ldots($	e5			e4			8	2	3
5: $\ldots N$		r6	r6		r6	r6			
6:$\ldots E+$	e5			e4				9	3
7: $\ldots T*$	e5			e4					10
8: $\ldots(E$		e6			e11				
9: $\ldots E+T$		r1	**e7**		r1	r1			
10: $\ldots T*F$		r3	r3		r3	r3			
11: $\ldots(E)$		r5	r5		r5	r5			

Calcul des états

Les états sont des ensembles de *règles marquées*

$$x \rightarrow u.v$$

correspondant à la situation: on a déja u en sommet de pile et on attend d'empiler v pour réduire uv en x.

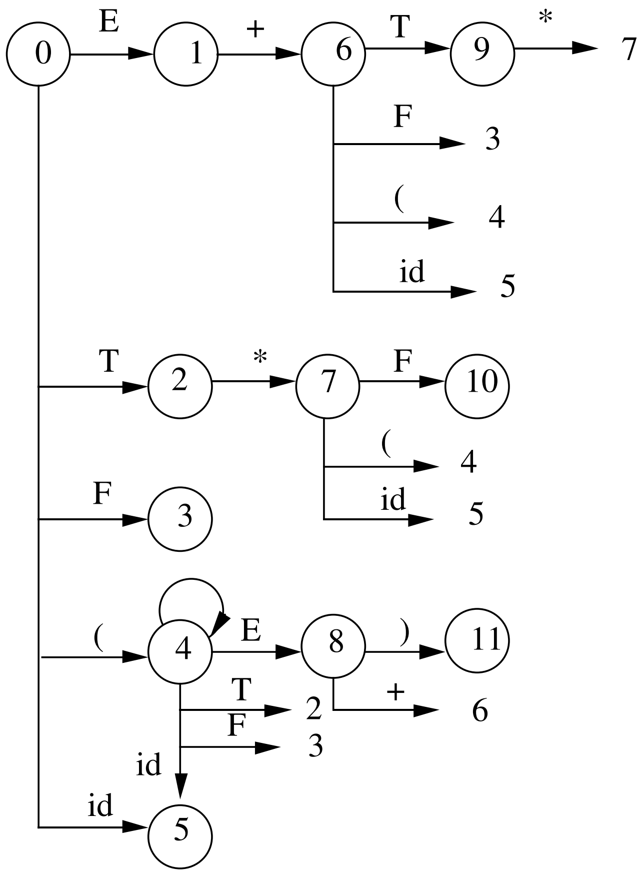
Les transitions sont de deux sortes:

1. empilement: passer de $x \rightarrow u.yv$ à $x \rightarrow uy.v$ avec lecture de y.
2. réduction: passer de $x \rightarrow u.yv$ à $y \rightarrow w$ sans lecture de symbole.

Pour chaque état, on représente seulement le noyau (avant de faire les ϵ-transitions). On ajoute la règle $E' \to E\$$. On obtient, pour la grammaire E, T, F les 12 états :

$$
\begin{array}{lll}
0 : E' \to .E\$ & 1 : E \to E. + T & 2 : \begin{cases} E \to T. \\ T \to T. * F \end{cases} \\
3 : T \to F. & 4 : F \to (.E) & 5 : F \to N. \\
\\
6 : E \to E + .T & 7 : T \to T * .F & 8 : T \to T * F. \\
9 : \begin{cases} E \to E + T. \\ T \to T. * F \end{cases} & 10 : T \to T * F. & 11 : F \to (E).
\end{array}
$$

L'automate LR



Analyse LR(1)

On prend cette fois dans l'état la lettre attendue comme lookahead au moment de la réduction. Les états sont des ensembles de couples

$$(x \to u.v, a)$$

avec les deux sortes de transitions:

1. empilement: sans changer de lookahead
2. réduction: on passe de $(x \to u.yv, a)$ à $(y \to w, b)$ avec $b \in \text{Premier}(va)$

Exemple

Grammaire des affectations en C:

$$\begin{array}{lcl} S & \rightarrow & L = R \\ S & \rightarrow & R \\ L & \rightarrow & *R \\ L & \rightarrow & \mathbf{id} \\ R & \rightarrow & L \end{array}$$

La grammaire n'est pas $SLR(1)$ car, si analyse **id**= **id**, on obtient:

PILE	DONNEE
	id=**id**
id	= **id**
L	= **id**

On a un conflit empilement/réduction car on peut réduire par la règle $R \rightarrow L$ puisque $=$ est dans Suivant(R)

Analyse LALR(1)

C'est la méthode employée par YACC. Elle est intermédaire entre $SLR(1)$ et $LR(1)$:

1. On calcule les états LR(1) et les transitions
2. On les regroupe sans mémoriser le lookahead
3. on utilise leur origine pour résoudre les conflits

Par exemple sur la grammaire précédente: l'état 2 provient de l'état

$$\begin{array}{rcl} S & \rightarrow & L. = R, \$ \\ R & \rightarrow & L., \$ \end{array}$$

et donc, sous $=$ il faut empiler et non réduire.

Exemple

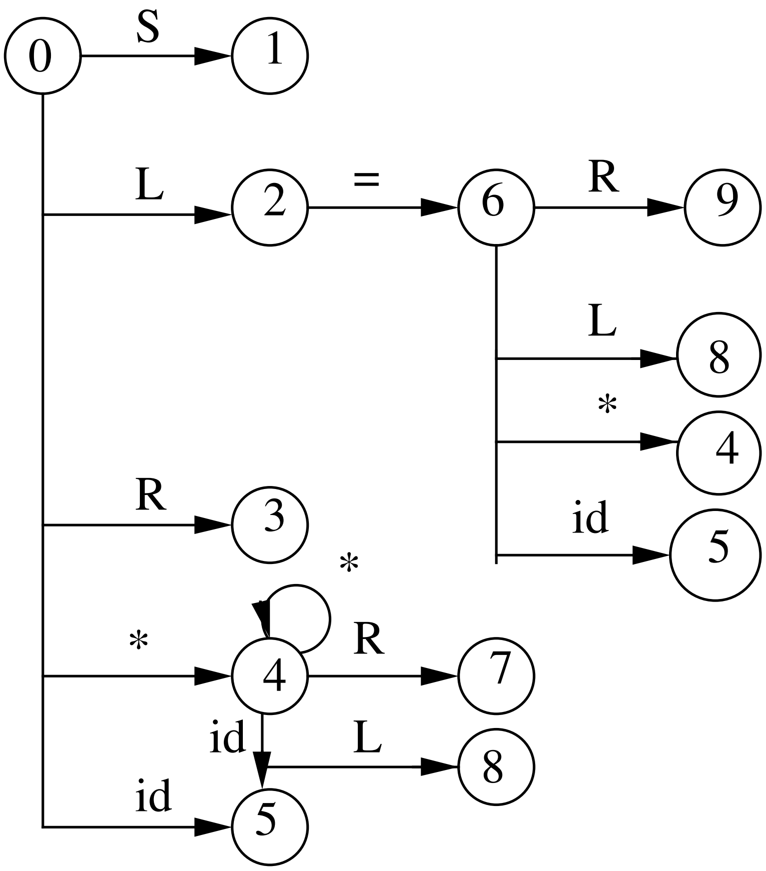


Figure 7: L'automate LR(0)

Table d'analyse

ETAT	terminaux				non-termin.		
	=	*	id	$	S	L	R
0		e4	e5		1	2	3
1				ok			
2			r5				
3				r2			
4		e4	e5			8	7
5	r4			r4			
6		e4	e5			8	9
7	r3			r3			
8	r5			r5			
9				r1			

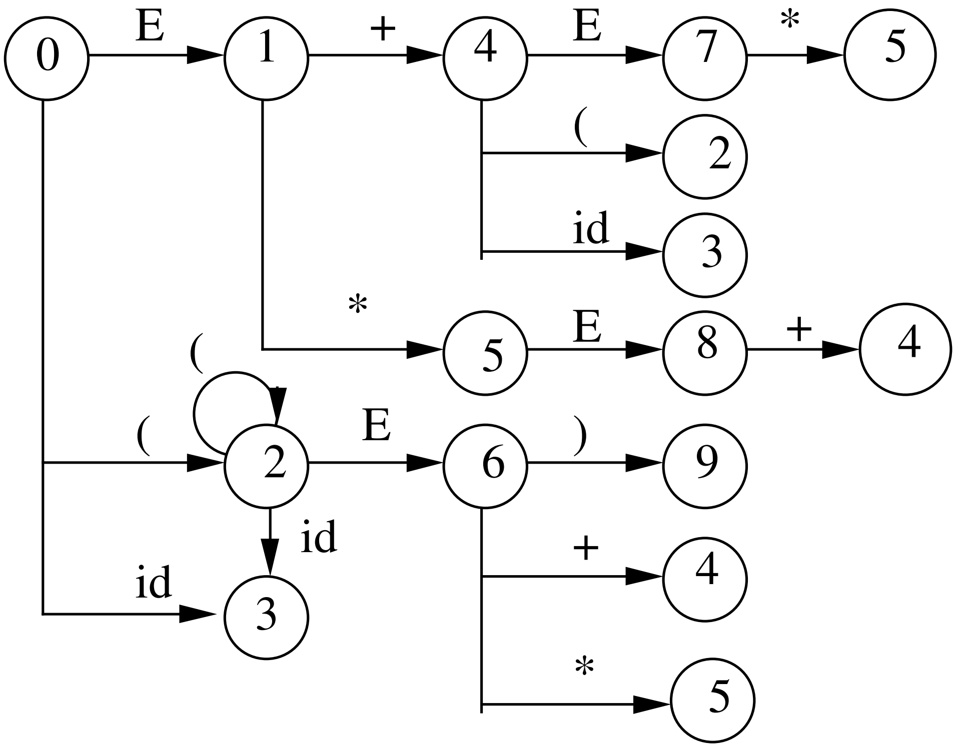
Utilisation de grammaires ambigues

On peut résoudre les conflits de l'analyse LR, même sur une grammaire ambigue en utilisant des règles de priorité. Ainsi, pour la grammaire:

$$
\begin{array}{lcl}
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & (E) \\
E & \rightarrow & \mathbf{id}
\end{array}
$$

on obtient une table d'analyse LR en résolvant les conflits par le choix de l'associativité de gauche à droite et la priorité de $*$ sur $+$.

Automate LR



ETAT	terminaux						non-termin.
	id	+	$*$	(	)	\$	E
0	e3			e2			1
1		e4	e5			ok	
2	e3			e2			6
3		r4	r4		r4	r4	
4	e3			e2			8
5	e3			e2			8
6		e4	e5		e9		
7		r1	e5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Traitement des erreurs

Deux problèmes:

1. Emission des messages (diagnostic)
2. Redémarrage (traitement)

Stratégies de redémarrage:

1. mode panique: on écarte successivement les symboles de la donnée jusqu'à trouver un symbole de synchronisation (; ou **end** par exemple).
2. corrections: on cherche à corriger la donnée (difficulté si l'erreur s'est produite avant le point de détection).
3. règles d'erreur: on ajoute à la grammaire des constructions incorrectes avec leur traitement.

Implémentation en analyse LR

1. mode panique: Le traitement consiste à dépiler jusqu'à trouver un état s qui ait une transition sur un symbole non-terminal A. On avance ensuite sur la donnée jusqu'à trouver un terminal a qui peut suivre A. On empile ensuite l'état (s, a) et on reprend l'analyse. Par exemple A peut être $inst$ et a le terminal ';'.

2. correction: Les appels aux procédures de traitement d'erreurs sont incorporés à la table d'analyse.

Exemple

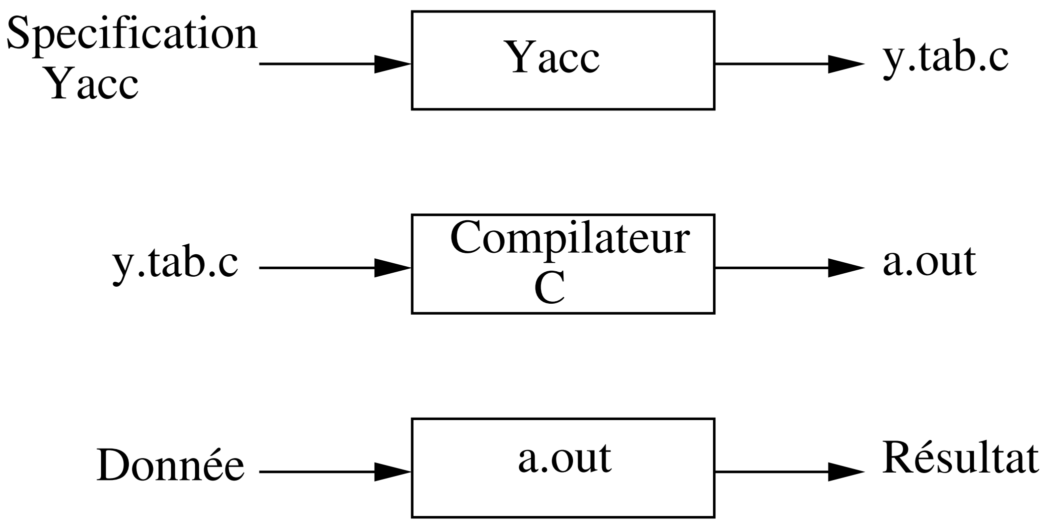
ETAT	terminaux						non-termin.
	id	+	*	(	)	$	E
0	e3	1	1	e2	2	1	1
1	3	e4	e5	3	2	ok	
2	e3	1	1	e2	2	1	6
3	r4	r4	r4	r4	r4	r4	
4	e3	1	1	e2	2	1	8
5	e3	1	1	e2	2	1	8
6	3	e4	e5	3	e9	4	
7	r1	r1	e5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3		r3	r3	

Erreurs:

1. empiler 3 (i.e. un **id** imaginaire) et émettre:‘opérande manquant’
2. enlever) de la donnée et émettre: ‘parenthèse ouvrante en trop’
3. empiler 4 et émettre: ‘opérateur manquant’
4. empiler 9 et émettre ‘parenthèse fermante manquante.

Constructeurs d'analyseurs syntaxiques

On peut utiliser des logiciels pour construire un analyseur syntaxique.



Yacc utilise la méthode LR (en fait LALR(1)) pour construire un analyseur.

Forme générale

Forme générale d'un programme source Yacc :

```
declarations
%%
regles de traduction
%%
routines C annexes
```

Les déclarations comprennent des déclarations C (entre `%{` et `%}`) et des déclarations de lexèmes.

Les règles de traduction sont de la forme:

```
<partie gauche> : <alt1>  {action1}
                | <alt2>  {action2}
               ...
                ;
```

où les actions sont des instructions C portant sur les attributs des variables ($$ pour la partie gauche puis $1 $2,... pour les variables de droite).

Calculette

```
%{
#include <ctype.h>
%}

%token CHIFFRE

%%
ligne   :       expr '\n'      {printf("%d\n",$1);}
        ;
expr    : expr '+' terme {$$ = $1+$3; }
        | terme
        ;
terme   : terme '*' fact {$$ = $1 * $3; }
        | fact
        ;
fact : '(' expr ')'      {$$ = $2; }
        | CHIFFRE
        ;

%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
```

```
        yylval = c - '0';
        return CHIFFRE;
    }
    return c;
}
```